Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 6
Today

- Review of concepts from last week
- Expanded types
- Control structures
Review of Concepts from Last Week

- Design by contracts
  - Preconditions, postconditions, class invariants
  - (Even partial) specification is useful
    - Test oracle, documentation
  - Use boolean expressions, instead of control structures
  - Never compare expanded type variables with Void
  - Use \texttt{implies} to specify conditional clauses in postconditions
  - Specify only what's been changed in postconditions
EXPANDED TYPES
What are reference and expanded types?

- **Reference types:** $s$ contains the address (reference, or location), of the object.

  **Example:**
  
  $s : \text{STATION}$

- **Expanded types:** $p$ points directly to the object.

  **Example:**
  
  $p : \text{POINT}$
How to declare an expanded type

➢ To create an expanded type, declare the class with keyword `expanded`:

```plaintext
expanded class COUPLE

feature -- Access

    man, woman : HUMAN
    years_together : INTEGER

end
```

➢ Now all the entities of type `COUPLE` will automatically become expanded:

`pitt_and_jolie: COUPLE`

Expanded types can contain reference types, and vice versa.
Objects of reference or expanded types

- Objects of **reference** types: they don’t point to any object when we declare them (they are \textit{Void})
  \[
  s : \text{STATION}
  \]
  We need to explicitly attach it to an object using a create instruction or an assignment.
  \[
  \text{create } s
  \]
  \[
  s := \ldots
  \]

- Objects of **expanded** types: they exist by just declaring them (they are never \textit{Void})
  \[
  p : \text{POINT}
  \]
  Feature \textit{default\_create} from \textit{ANY} is implicitly invoked on them
Custom initialization for expanded types

- Expanded classes are not creatable using a creation feature of your choice

  ```
  expanded class POINT
  create make
  feature make do x := 5.0; y := 5.0 end
  ...
  end
  ```

- But you can use (and possibly redefine) default_create

  ```
  expanded class POINT
  inherit ANY
  redefine default_create
  feature
    default_create
      do x := 5.0; y := 5.0 end
  end
  ```

Incorrect! But why?
## Equality and Assignment

<table>
<thead>
<tr>
<th>$a, b: C$</th>
<th>expanded class $C$ feature</th>
<th>class $C$ feature</th>
</tr>
</thead>
</table>
| $a := \ldots$  
$b := \ldots$ | feature  
Type1 $x$;  
Type2 $y$;  
end | feature  
Type1 $x$;  
Type2 $y$;  
end |
| $a = b$ | $a.x = b.x \text{ and } a.y = b.y$ | Do $a$ and $b$ point to the same object? |
| $a.is\_equal(b)$ or $a \sim b$ * | $a.x = b.x \text{ and } a.y = b.y$ | $a.x = b.x \text{ and } a.y = b.y$ |
| $a := b$ ** | $b$ stays unchanged;  
$a = b$ | $b$ stays unchanged;  
$a = b$ |
| $a.is\_deep\_equal(b)$ | $a.x.is\_deep\_equal(b.x)$ and  
$a.y.is\_deep\_equal(b.y)$ | $a.x.is\_deep\_equal(b.x)$ and  
$a.y.is\_deep\_equal(b.y)$ |

* Behavior can be redefined  
** The same semantics applies also to argument passing
True or False?

class VECTOR ...

```
1.0
2.0
(VECTOR )
```

1.0
2.0
(VECTOR )

```
a = b ?
True
```

```
a = b ?
False
```

```
1.0
2.0
(VECTOR )
```

```
1.0
2.0
(VECTOR )
```
True or False?

expanded class *POINT* ...

<table>
<thead>
<tr>
<th></th>
<th>1.2</th>
<th>5.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(POINT)</td>
<td></td>
</tr>
</tbody>
</table>

expanded class *COUPLE*

a = b?

a = b?

True

True
Their only privilege is to use manifest constants to construct their instances:

\[b: BOOLEAN\]
\[x: INTEGER\]
\[c: CHARACTER\]
\[s: STRING\]

... 
\[b := True\]
\[x := 5 \quad \text{-- instead of create } x.make_five\]
\[c := \text{‘c’}\]
\[s := “I love Eiffel”\]
Some basic types (*BOOLEAN*, *INTEGER*, *NATURAL*, *REAL*, *CHARACTER*) are expanded...

\[ a := b \]

\[ a = 3 \quad b = 5 \]

\[ a = 5 \quad b = 5 \]

... and immutable (they do not contain commands to change the state of their instances)...

\[ a := a.plus(b) \] instead of \[ a.add(b) \]

\[ a + b \]

Alias for *plus*
Strings are a bit different

Strings in Eiffel are not expanded...

\[ s : STRING \]

\[ s \text{.append ("very much!")} \]

... and not immutable
CONTROL STRUCTURES
Structured programming

- In **structured programming** instructions can be combined only in three ways (constructs):
  
  1. **sequential composition**
     - Compound
     - s_1
     - s_2
     - sequential composition

  2. **conditional loop**
     - Condition
     - True
     - False
     - s_1
     - s_2
     - conditional

  3. **loop**
     - True
     - False
     - s

- Each of these blocks has a single entry and exit and is itself a (possibly empty) compound
Control Structures in Eiffel

- \( c \) is of type BOOLEAN, not INTEGER or others
- \( s_* \) can be empty statement
- \( \text{exp} \) is of type INTEGER or CHARACTER
- \( \text{const}_* \) is a constant of INTEGER or CHARACTER
Programming Task: Pig Latin Translator

- Write a program that translates normal text in English into Pig Latin. Pig Latin translates words in the following way:
  - words that start with consonants have the consonants stripped and added as a suffix along with the string “ay”;
  - words that start with a vowel are appended with the string “ay”.

**Sample input:** This is a test for Pig Latin Translator

**Sample output:** Is-thay is-ay a-ay est-tay or-fay Ig-pay Atin-lay Anslator-tray

---

**STRING**
- is_empty: BOOLEAN
- item (i: INTEGER): CHARACTER
- has (c: CHARACTER): BOOLEAN
- split (a_separator: CHARACTER): LIST [STRING]
- append_character (c: CHARACTER)
- append (s: STRING)
- substring (start_index, end_index: INTEGER): STRING

**CHARACTER**
- is_upper: BOOLEAN
- as_lower: CHARACTER

**LIST[STRING]**
- start
- after: BOOLEAN
- forth
- item_for_iteration: STRING
~ END ~